Aalto University
School of Science and Technology
Faculty of Information and Natural Sciences
Degree programme of Computer Science and Engineering

Kalle Aaltonen

Mutation Testing in Automatic Assessment of Software Testing Skills

Master's Thesis
Espoo, May 24, 2010

Printing date: May 24, 2010

Supervisor: Professor Lauri Malmi, Helsinki University of Technology
Instructor: Petri Ihantola M.Sc. (Tech), Helsinki University of Technology

Aalto University
School of Science and Technology
Faculty of Information and Natural Sciences
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Kalle Aaltonen

Title of thesis:

Mutation Testing in Automatic Assessment of Software Testing Skills

Date: May 24, 2010 **Pages:** 6 + 59

Professorship: Software Technology **Code:** T-106

Supervisor: Professor Lauri Malmi

Instructor(s): Petri Ihantola M.Sc. (Tech)

Introductory programming courses rely heavily on using programming assignments to help teach students the basics of developing software. There exists a need to assess these assignments automatically to reduce the strain caused by manual inspection on limited course staff resources.

Learning to program includes software testing. Traditionally automatic assessment systems have relied on code coverage metrics to assess the test suites generated by the students. This work demonstrates weaknesses of this approach and proposes mutation testing as an alternative. Mutation analysis tool Javalanche is evaluated and used on actual coursework in the university's programming courses. The results are analyzed quantitatively and qualitatively to demonstrate the strengths and weaknesses of the new approach.

Keywords: thesis, mutation testing, programming exercise
Language: English

Aalto-yliopisto
Teknillinen korkeakoulu

Aalto-yliopisto
Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Kalle Aaltonen	
Työn nimi:	Mutation Testing in Automatic Assessment of Software Testing Skills	
Päiväys:	24. toukokuuta 2010	**Sivumäärä:** 6 + 59
Professuuri:	Ohjelmistotekniikka	**Koodi:** T-106
Työn valvoja:	Professori Lauri Malmi	
Työn ohjaaja:	Diplomi-insinööri Petri Ihantola	

Ohjelmoinnin peruskurssit ovat voimakkaasti riippuvaisia ohjelmointiharjoituksista opettaessaan oppilaille ohjelmistokehityksen perustaitoja. Näin on syntynyt tarve arvioida näitä harjoituksia automaattisesti, jotta voitaisiin vähentää käsin tehtävän tarkastuksen taakkaa kurssihenkilökunnalta.

Ohjelmistotestaus on osa ohjemoinnin oppimista. Perinteiset automaattiset arviointijärjestelmät käyttävät koodikattavuutta opiskelijoiden testien arviointiin. Tämä työ näyttää tämän lähestymistavan heikkouksia ja ehdottaa vaihtoehdoksi mutaatiotestausta. Javalanche-mutaatiotestaustyökalua arvioidaan, ja sitä käytetään analysoimaan opiskelijoiden harjoitusten vastauksia korkeakoulun ohjelmoinnin kursseilta. Näitä tuloksia arvioidaan kvalitatiivisesti ja kvantitatiivisesti näyttämään uuden lähestymistavan heikkouksia ja vahvuuksia.

Avainsanat:	diplomityö, mutaatiotestaus, ohjelmointiharjoitus
Kieli:	englanti

Contents

Abbreviations

ATDD	Acceptance test driven development
CAA	Computer aided assessment
MT	Mutation testing
TDD	Test-driven development

Chapter 1

Introduction

As the field of software engineering matures, software testing skills are becoming ever more important. This presents challenges in computer science education to equip the students with the testing skills needed. Computer science is a very practical field, and writing simple computer programs is a very natural way of learning to program. This software can be submitted in an electronic format to the educator and graded automatically. If the student also generates unit tests for program, it would be helpful to be able to grade it as well. This thesis work will address the problem of grading student generated test suites using mutation analysis. Currently the most popular metrics with which grading is performed are coverage metrics. We will demonstrate that mutation analysis offers superior capabilities in identifying weak test suites, but the details of fully integrating mutation analysis system into a current computer assisted assessment system are outside the scope of this thesis.

Mutation analysis is performed on a software by seeding simple programming errors into it. These "mutants" are generated systematically in large quantities and the examined test suite is run on each of them. The theory is that the test suite that detects more generated defective programs is better than the one that detects less of them. In theory any semantically significant modification of program source code should alter the behaviour of the program in a detectable way, but in practise many of the modifications don't affect the functionality of the program, which means that they should be selected carefully.

Mutation analysis was invented in the late 70's (DeMillo *et al.* , 1978), but so far it has failed to attract significant industry interest, mainly because of the high computational cost and the required human effort. Nonetheless there is active research and development taking place mainly in the academia. Mutation analysis tools are constantly improving, reducing computation costs and human effort re-

quired. This thesis will utilize one of the latest tools called Javalanche, a bytecode level mutation analysis tool for Java. (Schuler & Zeller, 2009) In addition, the programs created by students as their exercise work tend to be significantly smaller than real-world projects, which should make the application of mutation analysis easier in educational context.

1.1 Goals and scope

We will answer the following research questions:

Q1: Can mutation analysis be used to give meaningful feedback and grading on test suites included in programming assignment submissions?

Q2: Can the system presented in **Q1** be scaled to a handle large number of submissions in a course attended by hundreds of students?

Q3: Are there currently tools available that can be used to implement the system presented in **Q1**?

Question **Q1** was explored by examining actual coursework with student generated test suites and examining what kind of results mutation analysis can yield from it. This analysis was done both quantatively and qualitatively. Quantitive analysis was performed by running mutation analysis on a large number of test suites and comparing these results with other software metrics, specifically test coverage metrics. Qualitative analysis was done by examining a portion of these by experienced software testers to see whether or not the results of mutation analysis actually corresponded with expert opinion.

Question **Q2** was explored by observing the performance of the chosen mutation analysis tool on actual data. If the computational cost and time requirements are too high it can make the mutation analysis tool unfeasible for educational use in this context, as the student will have to wait for the feedback too long, or the course budget will be too strained by the need for computer time.

Question **Q3** was answered by conducting a survey of currently available mutation analysis tools. Even though mutation analysis is a relatively old idea, very few tools are available publicly today. Javalanche, a mutation analysis tool for Java developed by the Software Engineering Group at Saarland University, was chosen as our tool of choice for the constructive part of this thesis work.

This thesis will concentrate on the object oriented paradigm and more specifically on the Java programming language. Java is currently one of the most widely used

languages in industry. [1] It's also very popular in computer science education. Also practically all of the current tools for mutation analysis are for Java, but this is slowly changing.

Use of Java, and the object oriented paradigm in general, in computer science education is controversial, especially in introductory programming courses. The distributed control flow of object oriented programs and properly identifying objects in the problem domain are difficult subjects to learn, while many of the students attending these introductory classes have problems understanding the most basic computer program structures, such as loops. (Robins *et al.* , 2003)

1.2 Structure of the thesis

Chapter 2 gives an overview into computer aided assessment (CAA), as used in computer science education today. Different assessable features are identified, and the difference between dynamic and static assessment is explained.

Chapter 3 consists two sections: Section 3.1 gives an overview of software testing, its history, methods and tools. In section 3.2 we examine in greater detail how CAA can be utilized to develop students' software testing skills.

Chapter 4 consists of two sections: Section 4.1 describes the mutation analysis process, its history and the theory behind it, by presenting the major publications and other relevant related work. It focuses mainly on Java, and tools available to it. Section 4.2 focuses on the practical problems behind mutation analysis, and how it has been made sufficiently efficient to handle large-scale real world projects.

Chapter 5 presents the tools and the processes used to perform the experimental part of this work, where we want to run mutation analysis student generated test suites. Section 5.3 discusses what would needed to integrate this process into current CAA systems.

Chapter 6 gives an account of how Javalanche was used to perform mutation analysis on actual coursework in Helsinki University of Technology's course *Intermediate Course in Programming T1*. The results are analyzed and mutation analysis is shown to be superior to traditional methods in detecting weak test suites.

Chapter 7 summarizes the answers to the research questions **Q1**, **Q2** and **Q3**, introduced previously in this chapter. We will also provide observations on the Javalanche, and its specific suitability for our purposes. Potential applications of

[1] `http://langpop.com/` has an aggregation of programming language popularity with measurements from several sources. In the normalized results Java is currently leading with C and C++ holding second and third place.

this thesis's results and other possible future work are also discussed.

Chapter 2

Automatic programming exercise assessment

As computers become more ubiquitous in modern society, the demand for people programming them also increases. Traditionally programming has been the work of computer scientists specifically trained for the task, but today programming skills are taught to pupils in other technology disciples as well, who will need practical programming skills at some point of their careers. As a result, programming course sizes have become larger and put a greater strain on limited human resources.

Computer science is a very practical subject, and practical programming exercises are an effective way to ensure the student has learned practical programming skills, but the goals set by the teachers are not always achieved by the students. Grading and giving feedback on these exercises is labor intensive and time consuming when done manually. Automatic programming exercise assessment is the process of determining whether a programming exercise submitted by a student has been completed successfully, grading this and providing pedagogically valuable feedback on it. (Ala-Mutka & Järvinen, 2004) The exercise assessment is traditionally done by comparing the program's output with a reference, but it can also include more advanced analysis of the program's source code and its runtime behavior.

There are several problems with electronic submission and automated grading of programming assignments. Quality of the feedback can be poor, and the system may be too cumbersome to use. The system hardly serves its purpose if the student has to spend more time doing the actual submission than the exercise, which makes the usability of the system paramount. The system must also be secure to prevent cheating, as well as resistant to data corruption. Quality of the feedback can be enhanced with a semiautomatic process, where the system identifies potentially

interesting parts for manual inspection by the educators. (Ala-Mutka, 2005)

Advantages of automated grading are consistency, efficiency, thoroughness, and constant availability. Several persons grading submissions might judge an individual submission differently because of differing points of view or human bias. Automated systems on the other hand are deterministic, and will produce the same analysis time and again. Automated systems can work around the clock and their performance doesn't suffer from fatigue, as opposed to their human counter-parts. (Carter *et al.* , 2003)

The educators' expectation and attitudes toward computer assisted assessment (CAA) vary greatly. In 2003 an international survey was conducted among computer science educators about, and among other things, their perceptions of CAA and its usefulness. 64% of the respondents had experience with some form of CAA. It was generally agreed that CAA provides greater objectivity. The greatest point of disagreement among the people using and not using CAA was whether or not the CAA was more flexible to the students. Another point of disagreement was, that the persons who didn't have experience with CAA strongly disagreed with the idea that CAA systems can provide quality feedback or can test higher-order learning. The immediacy of feedback on the other hand was seen as positive thing by both groups. (Carter *et al.* , 2003)

Programming assignments may also include the student creating a test suite for the program, and the student would benefit from meaningful grading and feedback on it. Currently grading of the test suites is based on coverage metrics. The relationship between developing testing skills and automatic assessment systems is explored further in Section 3.2. In courses using and teaching test-driven development (TDD) test suite quality is especially important. (Edwards, 2003)

Students minimize their time completing the exercise and are prone to copy their work from their classmates, so mechanisms for detecting plagiarism have been developed. Also individually varying exercises can be used. (Ala-Mutka, 2005)

Other practical requirements for CAA include security and performance. The system should be secured against tampering by curious students. Since CAA is mostly used in large courses, the performance of the system can become an issue, especially when several students submit their work under an approaching deadline.

2.1 Automatic assessment for different features

The goal of the programming exercises is to give the student the ability to produce quality software. In this section we will describe a number of software quality fea-

tures, which can be assessed by automatic systems. Ala-Mutka, 2005 lists different features measured by current automatic assessment system, as illustrated in Figure 2.1. The features measured should be derived from the teaching goals of the course. Methods for assessing them can be split into two categories, dynamic and static assessment.

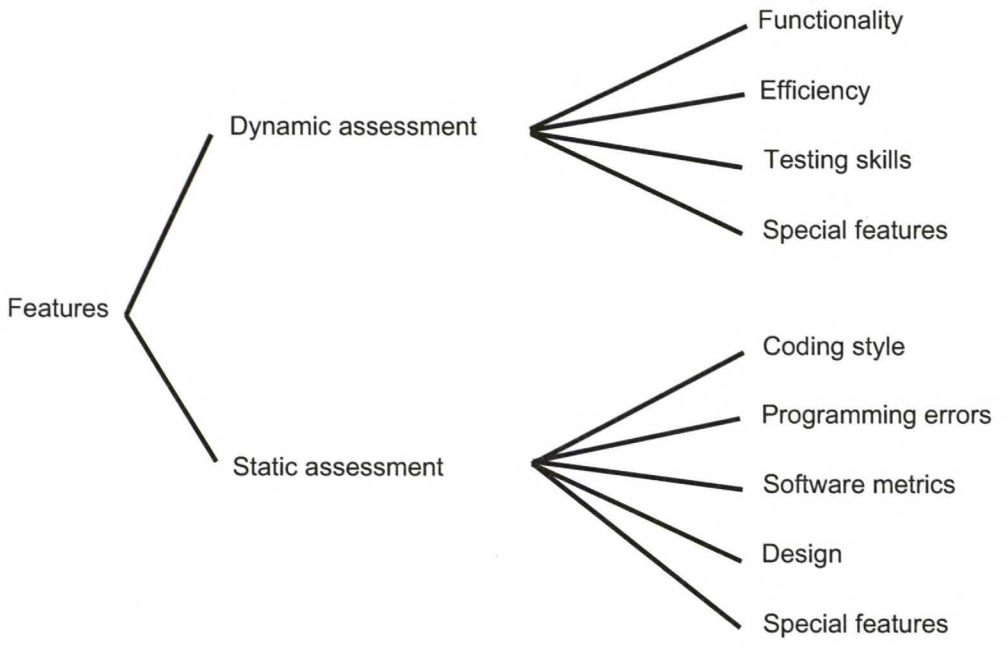


Figure 2.1: Taxonomy of software features covered by automatic assessment systems

2.1.1 Dynamic assessment

Dynamic assessment involves running the software and examining its output and runtime behavior. Compiling the program is a necessary first step. Compilation failures and warnings can be given as immediate feedback. The program submitted by the student can include memory management errors and infinite busy loops, and even code segments with malicious intent, all of which may cause the computer running the assessment process to perform poorly. These processes should always be sandboxed properly in order to prevent complete failure of the system.

Functionality

The functionality quality feature is whether or not the software performs the functions it was specified to do correctly. This is the most common feature assessed by

current systems. Correctness of the response can be examined by simply running the whole program with given input and matching its printed output with a pre-calculated result, or by running single methods in the program with a unit testing tool. If some of the test cases are given to the student with the exercise material, it can be thought of as (acceptance) test driven development ((A)TDD). If the exercise includes creating a graphical user interface (GUI), special tools are needed to simulate the human interaction.

Functionality is probably the first thing assessed by CAA systems, for example Web-Cat (Edwards, 2004), Assyst (Jackson & Usher, 1997), CourseMarker (Higgins *et al.* , 2003), etc.

Efficiency

Efficiency, as a software quality feature, means that the software satisfies its requirements without excessive resource utilization, and executing specific functions within their time limits. It can also mean reaching some real time constraints presented by the environment.

Efficiency is usually measured by executing test cases on the program and comparing the execution behavior with a model. Different aspects benchmarked can include execution time, CPU time, peak memory usage, I/O times, etc. Instead of executing the whole program, execution behavior of single methods can also be examined.

Efficiency is an assessable feature in several systems, including Web-Cat, Assyst, CourseMarker, etc.

Testing skills

How well the student has tested his software. This thesis focuses on this feature. Students should be encouraged to perform their own testing on the software instead of using the assessment system to do it for them. If the software is thoroughly tested before submission, it is probably reasonably **testable**.

Testability can be assessed by specialized software tools, such as Testability Explorer for Java. (Hevery, 2008) It assesses testability by determining how many of the class variables are injectable. In order for the tester to inject his own mock classes into the system, the software structure must support it. They also calculate the number of mutable global states. They feel that these global variables present hidden dependencies and offer poor isolation for the tests. This only applies to non-constants. (Hevery, 2008)

The student generated test sets are assessed by Web-CAT, which uses coverage metrics, as well as Assyst, which uses statement coverage.

Special features

In addition to the dynamically assessed features listed above, the exercises can include language or domain specific constraints. For example, a piece of software to be implemented without dynamic memory allocation, or without using certain other features of the language or environment. These can be caught in runtime by overriding the functions in question with sentinel functions, which cause the software to fail the assessment.

2.1.2 Static assessment

Static code analysis is examining the program without executing it, as opposed to dynamic analysis which happens at runtime. This examination is done by automated tools, where as *code reviews* are form code examinations performed by humans. Code reviews are the most basic way of assessing programming assessments, as it doesn't require compilation, or even a computer if source code is in a printed format. Today most exercises are submitted in electronic formats.

Numerous software metrics can be generated with static analysis; program size in lines of code, number of classes, methods, their respective size, comment density, as well as more advanced metrics such as cyclomatic complexity.

Coding style

Requirement for coding style analysis is that the program is syntactically correct. Easiest way to determine is to use a compiler for compilable languages. Modern compilers are very good at finding potentially problematic parts of the program and warning about them (unused variables, implicit casting, etc). They can be information sources for assessing coding style.

In addition to compiling and providing the required functionality, computer programs are often also read by other persons. Clearly there are code style considerations besides the program being technically correct. Generally the easier it is for another person to understand and extend the program, the more valuable it is. **Understandability**, as a software quality attribute, means that the software is easy to approach, it is easy to understand how it works and was implemented. This means that variables and classes are defined and named reasonably. Program statements should be descriptive and easy to read. Methods should be short and

perform a clearly defined task. Understandability can also be improved by adding comments. Assessing the code commenting is particularly challenging. The logical first step would seem to be to award the student for any comments in the source or penalize those without them. Howles, 2003 gives an account of such a system; once the students realized that the amount of comments affected their score, they started adding comments which contained simple gibberish. Automatically assessing the *quality* and the relevance of the student generated comments would at very least require natural language understanding.

Coding style also affects **maintainability** and **testability**. Maintainable software is easy to modify or expand and has clearly defined interfaces and components, and is otherwise modular in its structure. Testability measures the effort required to effectively test the software. Testable software avoids complex structures which make the software hard to test. This is accomplished by splitting the program in subroutines and components which can be tested individually. Long methods are hard to understand and test well.

There are also language specific style considerations, such as the code conventions presented in Java language specification. (Gosling *et al.* , 2005) These include variable and class naming, code indentation, etc. Software should be **consistent** with its terms and notations. It should follow a single programming style and not deviate from it.

Actual tools for assessing coding style include style++ for C++ (Ala-Mutka *et al.* , 2004), and the Checkstyle[1] plug-in for Web-CAT.

Programming errors

A significant portion of programming errors are difficult to detect with dynamic analysis, i.e. running test cases on it. Many of these problems appear sporadically or after the program has been in execution for a prolonged period. Typical examples are memory allocation errors, or concurrency related problems. Programming errors can also include failure by the student to embrace the programming paradigm of the language in question, e.g. a student with a background in imperative languages might not write functional programs in the style that is expected, even if the resulting program functions correctly.

There are many static analysis tools for finding potential problems and errors in source code, the most famous of them being **lint**[2].

[1] `http://checkstyle.sourceforge.net/`

[2] Lint first appeared in V7 of the UNIX operating system in 1979.

Software metrics

Numerous metrics can be derived from computer programs. If these metrics work as measurements that characterize computer programs, they can used in the assessment. These metrics can usually be easily calculated, but if the measurements are used in the grading, they must support the educational goals of the course.

Typical metrics include cyclomatic complexity, program size (lines of source code, number of files/methods/classes/modules) and amount of comments. Cyclomatic complexity is the measure of independent paths through the program. Both these metrics can indicate whether or not the student's program is too complicated or large.

Software should be **concise**. It shouldn't contain redundant or unnecessary information. This also includes the absence of redundant or otherwise similar code. Code shouldn't also be overly verbose.

One open source tool for extracting software metrics from the code is Testability Explorer (Hevery, 2008), uses software metrics to give feedback about the testability of the software. Assyst computes the cyclomatic complexities of the software. (Jackson & Usher, 1997)

Design

Often the exercise requires the student to conform to a certain interfaces or structural requirements, which are usually checked automatically. The exercise might concern certain design patterns, the presence of which ascertained through static analysis. Antoniol *et al.* , 2001 presented a tool that recognizes common design pattern from the programs source code.

Special features

Design requirements can also include denying the student access to certain language features or libraries when completing the assignment. At the simplest these can be detected by examining the source for significant keywords, examples of these might be `malloc` and `free`.

Plagiarism is also included in this category. Since programming exercises are on electronic format, they are trivially shared among the people attending the course. Even though the trivial case where the plagiarized response in identical to original is easy to detect, the problem becomes much harder when simple changes are applied to the code. Several tools have been developed for discovering the structural similarities

between two programs, such as Moss[3] and JPlag[4].

2.1.3 Other attributes

There are numerous other attributes, besides the ones covered in the previous section, that are associated with software quality.

Reliable software is resistant to exceptional conditions and can run for extended periods of time providing the specified functionality. Malformed input and malicious user interaction should be withstood without a catastrophic failure. Reliability is typically assessed with stress testing, where system is exposed to testing beyond normal operating conditions, where its behavior, robustness and availability is assessed.

Other software quality attributes also include **security**, **portability** and **usability**. Many of these presented attributes have no clear definition and are subjective. These are very hard to judge automatically.

[3] `http://theory.stanford.edu/~aiken/moss/`
[4] `https://www.ipd.uni-karlsruhe.de/jplag/`

Chapter 3

Software testing

As long as there has been computer software, there has also been a need to demonstrate its correctness and fitness for the intended purpose. Obvious solution to this would seem to be formal verification where every part of the software is proven with formal methods of mathematics. This is usually done with model checking or logical inference.

Model checking is the process of comparing every state of the model against the specification. In many situations the state space of the software is so large that model checking is impractical. Methods for reducing the size of the state space have been developed but still very few non-trivial programs are verified in this way.

Logical inference is using formal mathematical reasoning to prove with software with interactive theorem proving systems. These can only be partially automated.

While formal verification is very widely used in the computer hardware industry, it has not enjoyed much success in the software industry, most likely due to its prohibitive costs.

An alternative method for demonstrating software fitness is software testing. Software testing can be formally defined as: (IEEE 610.12, 1990)

1. The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

2. The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.

Software testing can be described as an activity directed at finding faults (defects, bugs) in the software. These faults are caused by errors made by the programmer.

Software testing can demonstrate the existence of these errors manifesting themselves as faults when executed under certain conditions.

In 1972, Dijkstra famously declared that "program testing can be used to show the presence of bugs, but never their absence". (Dijkstra, 1972) Despite this huge amount of research and industry effort has been directed at software testing, and today it remains the primary method of demonstrating software fitness.

3.1 Overview of software testing

In this section we present some of the current techniques and methods of software testing.

3.1.1 Testing methods

Software testing can be divide into two categories based on how it's done; black and white box testing. They describe the level of visibility that is required for the tester working on the system.

Black box testing

Black box testing treats the software as a closed system, and the tester doesn't have access to internal workings of the system. Only the inputs, outputs and general function are known, but its content and implementation are unknown or irrelevant. Typical black box testing techniques include: (IEEE 610.12, 1990)

- **Equivalence partitioning**, where the test data is divided into partitions, which should be covered by a test case at least once.
- **Boundary value analysis** test the edges of equivalent partitions, which are common locations for errors.
- **All-pairs testing** is a combinatorial testing method, where all pairs of input parameters are tested. This is a compromise between testing all possible combinations of the input parameters.
- **Fuzz testing** inputs invalid, unexpected or random data into the program and observes its behavior. Failures of the system can lead to discovery of defects.
- **Model-based testing** is performed having a possible incomplete model of the system and comparing its behavior to the tested software.

- **Traceability matrix** is document where requirements of the system and the test cases are displayed in a matrix, which can be used to demonstrate that all the listed requirements are covered by the test set.
- **Exploratory testing** emphasizes creativity and freedom of the tester to create test cases. This can also be called *ad hoc testing.*
- **Specification-based testing** consists of test cases directly derived from the specification document. Without a very detailed specification document this alone is usually insufficient.

Some literature draws a distinction between black box and grey box testing. Grey box testing means having access to the internal data structures and algorithms when designing the testing, while using black box testing techniques.

White box testing

White (or glass) box testing is the opposite of black box testing, where the internal structure and implementation is known and it is treated as such, which means implementation specific issues can also be tested. Typical white box testing techniques include: (IEEE 610.12, 1990)

- **API testing** is systematically testing the public and private APIs (Application programming interface) of the application.
- **Code coverage testing** is creating test sets to satisfy a specified code coverage criterion.
- **Fault injection** is done by introducing errors into the application and examining whether or not the test set detects them.
- **Mutation testing** is similar to fault injection, where the possibly faulty programs are created in large quantities by a mutation testing tool in a systematic fashion. Quantitative analysis between the detected and undetected defective programs can be used as to measure of test quality.
- **Static analysis** is examining the source code of the program by for example code reviews.

3.1.2 Testing levels

Software testing is often divided into several levels that typically match different phases of the software development process. This is often referred to as the V-model, as illustrated in Figure 3.1.

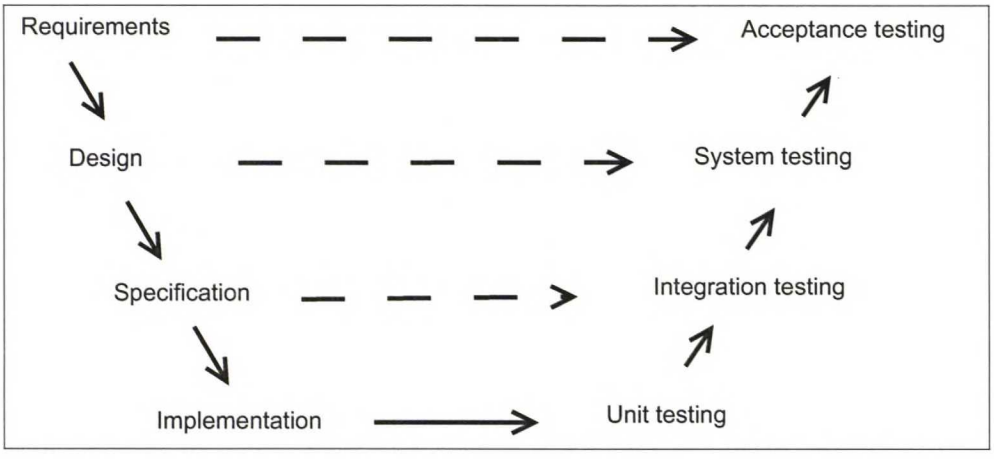


Figure 3.1: Software engineering V-model

Lowest level of testing is unit testing, where the individual identifiable components are tested in isolation. This is often done in conjunction with the implementation and covers the basic functionality of the component to be implemented. If the unit tests are done before the actual implementation, the resulting process is called test-driven development.

Integration testing tests the interaction of several basic components, and is derived from the software specification. They focus on the functionality and structure of the system.

System testing tests the system as a whole, with all the components integrated together. This level of testing usually includes non-functional attributes of the system (see Subsection 3.1.3), such as performance, usability, etc.

Acceptance testing is usually the highest level of testing. It is done to ensure that the software meets its customer requirements. Acceptance test phase is also used by the customer determine whether to accept the system or not.

3.1.3 Functional versus non-functional testing

Functional testing is motivated by the system meeting its explicit functional requirements. Non-functional requirements include usability, security, scalability, performance.

Usability can be defined as the ease with which the user can employ the tool to achieve a particular goal. (IEEE 610.12, 1990) Usability testing is usually performed by persons who represent the actual end users of the software. The test subject is asked to performed a series of tasks the software has been designed to use perform, and the interaction of the user and the system is observed. User-centered design paradigm is a design philosophy where usability is the center of the software design throughout the process.

Security, as a non-functional system requirement, pertains to the information security of the system. This requirement is largely dependent on the sensitivity of the data contained, the environment in which the software is to be run, etc.

Scalability is the capability of a software system to be upgraded to accommodate increased loads. This is usually tested with load testing, where the system is tested by putting a certain level of demand on the system and measuring its performance.

Performance requirements can include execution time and memory requirements. It's is usually done by running the software in a specific environment and input, and verifying its resource usage.

3.1.4 Test automation

Manual testing is labor intensive, and this burden has been alleviated with a myriad of tools. Software test automation is the use of specifically designed software tools to control the execution of tests, comparing the output with the expected output, and other control and reporting functions. The automated tests can be either code-driven or GUI (Graphical User Interface) testing. Code-driven testing involves using the available interfaces of the software various inputs and validating this output. GUI testing is done by simulating the interaction of a user with the software. This can be random interaction with the GUI components available with the intent of driving the system into an unexpected state, or sophisticated scripts derived from the specification document.

Several frameworks for test automation exist:

- **Data-driven testing** is a test automation framework where the test logic and data are stored separately and the same test logic is reused with several data sets. These data sets are usually stored in a database.
- **Modularity-driven testing** is used by creating small independent tests that represent the different modules and functions in the software. This is a very simply and popular approach to test automation.

- **Keyword-driven testing** (also table-driven testing or action-word testing) is testing methodology, where the testing is separated into two stages; planning and implementation. Keywords are simple actions or objects that are defined during the planning stage. After the tests have been implemented for the keywords, a driver is used to execute the test suite.

- **Model-based testing** involves using a model that at least partially describes the system under test. The test cases are derived from this model. It can be seen as a form of black box testing.

- **Hybrid testing** frameworks are what most frameworks eventually transform into. They utilize different aspects of the for previously mentioned frameworks.

3.1.5 Test adequacy criterion

As stated before, software testing is unable to show the absence of programming errors in the software, only their presence. After Dijkstra's made his famous statement, it was quickly discovered that th central question of software testing is the test criterion, "what is an adequate test set?" (Goodenough & Gerhart, 1975) Goodenough and Gerhart present in their paper that test adequacy criterion should be a predicate of how "thorough" the test is. A thorough test's successful execution should imply that there are no errors in the tested program.

Several test adequacy criteria have been used:

- **Statement coverage** is calculated by examining how many of programming language statements in the program are executed by the test set. A test set is considered adequate in respect to statement coverage criterion, when all the statements of the program are executed at least once by the test set. Statement coverage can also be used as a metric, when the percentage of executed statements is calculated.

- **Branch coverage** is similar to statement coverage. The difference is that instead of single statement, all the flow control structures in program are examined. The ratio of executed branches to the total number of branches is branch coverage.

- **Path coverage** requires that all the possible execution paths through the program are enumerated and executed at least once during testing. In most non-trivial programs the number of possible execution paths is not enumerable, which makes it impractical in most cases.

- **Mutation adequacy** is measured by seeding large number of defects into the tested program and examining whether or not the test set detects them. Mutation testing is covered in more detail in Chapter 4.

While various test adequacy criteria have been developed, it's still an open discussion whether or not these criteria actually capture the true notion of test adequacy; what is the fault detection ability of the test set, and what is the dependability of the software that has passed this test? (Zhu *et al.* , 1997)

3.2 Software testing skills

Industry opinion is that recent graduates have poor understanding of testing skills relating to software engineering, while defective software causes losses in productivity estimated to around $100 billion annually. (Edwards, 2003) They believe that even introductory level software courses should expose the student to testing their own, often trivially small, exercise submissions. They should be tested with unit tests, which introduce the students to test automation and its tools early in their programming careers.

This view is also supported by Howles, 2003, who did a student survey, which discovered that only 39% of the students performed static testing on their software before submitting it for grading. Majority of the students didn't perform any kind of unit testing. She also speculates reasons behind this behavior: The exercises normally serve no other purpose for them, except to award them for the work done. Once the assessment system indicates that the submission has been completed with grade that matches the student's expectation, it becomes useless and unneeded. No regression testing is needed, which makes any test suite worth even less. It's only once the student becomes a part of a collaborative effort to produce a larger piece of software that attention to testing becomes more important for them. Then there is no way to receive immediate feedback, whether or not their work of acceptable quality and their peers depend on it to fill its role, which makes it easier to justify the effort spent on testing the software. This is the fundamental reason why better automatic assessment methods than coverage metrics are needed for the test suite assessment; the student will simply augment the test set until sufficient coverage is reached and stop, regardless of what the test suite quality is. The perfect coverage score can be achieved without specifying a single assertion about the programs behavior. (Howles, 2003)

Automatic assessment systems present several problems for developing testing

skills: Students are not usually rewarded or encouraged to perform their own testing. They focus on the correctness of the output as specified in the assignment and little else. At worst if the grading system has low latency and the student can do large number of submissions, the student can use it as his test suite. A trial-and-error approach can ensue, in which the student neglects all testing and uses the grading system for the testing. This tendency is understandable, if we assume that the average student will try to use the minimum amount of time to achieve their target grade. If using the assessment system as the sole testing tool is a more cost-effective way of achieving this goal, then it is rational for the student to neglect doing his own testing. In order to discourage this non-constructive trial-and-error approach certain design considerations should be taken when designing the assessment system, such as limiting the number of submissions, increasing the interval between them, or awarding the student for generating a test set.

3.2.1 Test quality

In order the award the student for generating a test set we need to examine the principles which make a good test set. Test suite coverage metrics can be used to find the portions of the software covered by testing. In addition to the test program executing a portion of the test, it needs to make sure that it behaves as expected. Typically these are assertions specified in the test cases. Assertions are predicates placed in the software or its test suite that developer thinks should always evaluate as true. The coverage metrics aren't affected by the presence of the assertions, which clearly makes them inadequate for assessing a test set. They can, however be very effectively used to find untested portions of the software. In Chapter 4 we present one possible technique for assessing the quality of the assertions.

Test quality is more than just code coverage and assertion quality: There are structural and style considerations to take into account when designing a test set, just as there are when designing the software itself. Several authors have presented guidelines for test design, as well as several anti-patterns (or *test smells*, see Table 3.1). Test smells are poorly designed tests, a concept introduced by van Deursen *et al.* , 2001. As the software evolves, the tests have to co-evolve with it, which increases the maintenance effort and cost. This means that any complex test cases with very high coupling to several software components can dramatically increase the maintenance cost of the software. The tests that severely hinder the modifications can render the test suite a cost-ineffective quality control mechanism in the end. (van Rompaey *et al.* , 2007)

Name	Description
Mystery Guest	The test depends on an external resource, i.e. test data file, database, external service. The test is no longer self contained.
Test Run War	The tests run fine when run by a single developer at the time, but stochastically fails miserably when run concurrently with another instance of itself. Usually related to the initialization and shared use of external resources.
General Fixture	Several tests reuse the same complex fixture. Any modifications to this existing large fixture can result in unexpected behavior in the other tests. Complex fixtures usually also have higher setup cost, which affects the performance of the test suite. These are typically huge `setUp()` methods in the xUnit family of tools.
Eager Test	The test checks several features of the class to be tested, which makes its purpose unclear. As unit tests also often serve as documentation, this should be avoided.
Assertion Roulette	The method has several assertions, but no explanation attached to some of them. Makes it more difficult to see which assertion has failed.
For Testers Only	Production code contains portions that are only used and needed by the test suite.

Table 3.1: Typical test smells

While some of these test quality attributes can be assessed automatically with static analysis, very little has been published about such work. These are also outside the scope of this thesis, as we're concentrating on assessing the quality and coverage of the assertions. We propose to use mutation analysis to assess the quality of the assertions, but it certainly won't a panacea for the difficulties of teaching software testing skills. We learned that coverage metrics can only find portions of the software that haven't been tested, but says nothing about how well the covered portions are tested.

Test quality of real world projects is of course finally dependent on business requirements of the software. The most important features should most thoroughly

tested and documented, as they generate the value of the project and the increased cost of testing and maintenance can be justified. In an educational context the "business requirements" the student is satisfying is getting enough points from the exercise to achieve his target grade.

Chapter 4

Mutation analysis

Mutation analysis is technique for determining the adequacy of a test suite. It consists of injecting faults into a computer program and examining whether or not a test suite will detect these faults. The theory is that, if the test suite fails to find the seeded fault, it's likely that it would fail to detect similar real defects in the code. On the other hand if the fault is detected, the test has demonstrated its quality. When tests are added or augmented so that the undetected faults are caught by the test suite, it's possible that real faults in the program surface that hadn't been detected by the earlier test suite.

Traditionally test suite adequacy is determined by different code coverage metrics (e.g. statement, branch, path coverage). Safety-critical applications may be required to reach 100% on some coverage metric. The coverage metrics are not however a good way to determine test suite adequacy. For example consider a Java method in Listing 4.1 and two alternative test suites for it in Listings 4.2 and ??:

Listing 4.1: Example method for calculating Fibonacci numbers

```
public static int fib(int n) {
    if(n <= 0)
        return 0;
    if(n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Both test suites achieve 100% statement coverage even though it's obvious that the test suite B is superior. We could show this by introducing an error into the program, for example by replacing `return 1` with `return 0`. Then the test suite

Listing 4.2: Fibonacci test suite A

```
assertTrue(fib(6) >= 0);
```

Listing 4.3: Fibonacci test suite B

```
assertEquals(8, fib(6));
```

B would detect this fault while test suite A wouldn't. If we were to generate large set of these faulty programs systematically and examine the results quantitatively we would have mutation analysis. However code coverage of the test suite can be calculated with a single execution of the suite, and can quickly and efficiently identify untested portions of the program and should not be dismissed.

An individual program with a single injected fault is called *a mutant.* Mutants are generated in large numbers and the ratio of detected mutants to all non-equivalent (see Subsection 4.1.2) mutants is called the *mutant score.* In an ideal situation this ratio is 100%, which means that all the changes in the program are detected by the test set. In this situation the test set is said to be *mutation adequate.* In the opposite end ratio 0% means that the test set failed to detect any of the changes made, which means that the test set is completely inadequate.

The first major publication about mutation analysis was in 1978. (DeMillo *et al.* , 1978) The injected faults are usually simple programming errors that are common in programming; using the wrong operator, wrong variable name, wrong method name with the same signature, etc. We will explain in Subsection 4.1.1 why finding most simple faults is believed to also reveal most complex faults.

Throughout the history of mutation testing the inefficiency of generating and eliminating mutants has been a barrier preventing it from reaching widespread use. Today advances in the field have made possible tools like Javalanche (Schuler & Zeller, 2009), which have been used to run mutation analysis on projects with over one hundred thousand lines of code in a few hours.

4.1 Mutation analysis process

Mutation analysis has two inputs; the program and its test set. First mutants are generated from the program. This is done using *mutation operators* (or mutation rules, mutation transformations in some literature), which are simple manipulations of the program. After a set of mutants has been created the test suite is run on each of them. If the test suite detects the program as faulty, the mutant is *killed.* If the test suite passes the mutant is alive and will be examined later. Ideally test cases are added so that all the mutants that are left alive are detected, and the process is repeated.

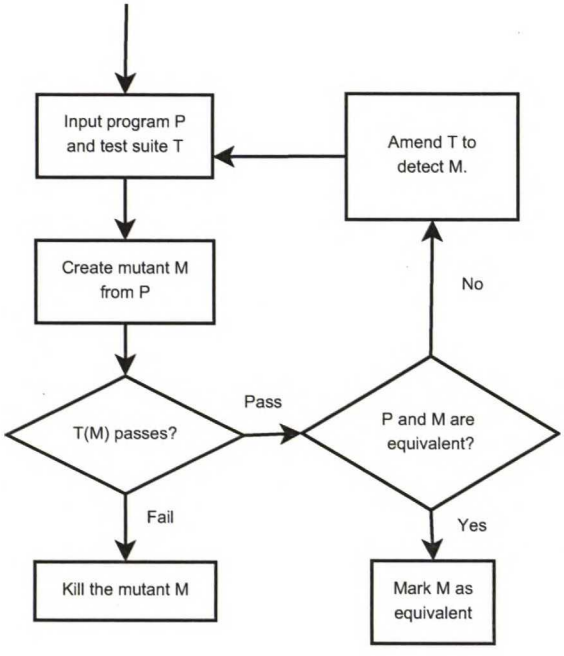


Figure 4.1: Mutation analysis process

The different phases of mutation analysis are described as a flow diagram in Figure 4.1. Process inputs are the program to be tested `P` and the test suite to be evaluated `T`. In the next phase mutants are generated from `P`, which is described in more detail in Subsection 4.1.1. Each generated mutant `M` is tested with `T`. If `M` fails in the testing this mutant is killed (Subsection 4.1.3). If the mutant passes the test suite, it is called a live mutant. Live mutants are examined in the next phase by hand and split into two categories; equivalent and non-equivalent mutants (Subsection 4.1.2). Tests are added to test suite `T` so that all the non-equivalent mutants are detected.

According to the fundamental premise of mutation testing, the less non-equivalent mutants are left alive, the less faults remain in the software:

> "In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault." (Geist *et al.* , 1992)

4.1.1 Mutant generation

Mutants are generated using mutation operators. Each operator is applied to the input program and the resulting mutants are stored for later examination. The

amount of mutants generated depends on the set of operations used and the program size.

Mutants can be generated by modifying program on different levels. It can be done on any level from machine code to interpreted languages with high abstraction level. Current mutation analysis tools for the Java language generate the mutants from the Java source code or the intermediary bytecode executed by the Java Virtual Machine, as illustrated in Figure 4.2.

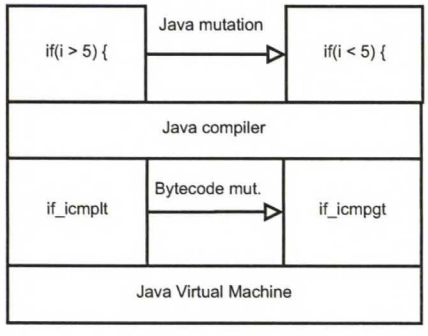


Figure 4.2: Mutant generation on the Java architecture

There are pros and cons to both source code and bytecode mutants:

- Each examined source code mutant has to be compiled, which is slow.
- Bytecode mutants are difficult to examine afterwards as it's not possible or straightforward to generate the Java source for the mutated bytecode.
- The compiler can eliminate dead code, which in theory can result in less equivalent mutants in source code mutants, e.g. if the mutation operation targets a part of the code that's deemed dead by the compiler, the resulting bytecode will be identical with the original.
- Some more advanced operators are significantly easier to implement in Java than in bytecode.

Mutation operations

Mutation operators modify the target program in some meaningful way. These are simple errors that could have been made by the developer. As the list of possible mutation operations on different languages is practically endless, we will focus on Java by presenting a μJava (or muJava). (Offutt, 2008) μJava's mutation operations

fall into to two distinct classes; method-level mutation operators and class mutation operators.

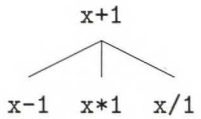
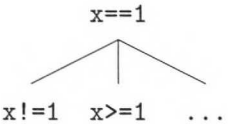
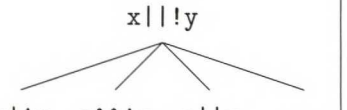
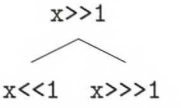
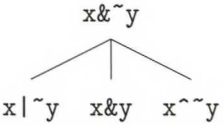
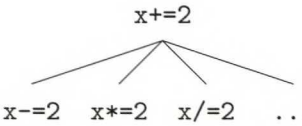
Name	Description	Example
Arithmetic operators	Replace, add, and remove unary and binary arithmetic operators (+, -, /, *, ++, --) for both integer and floating point operators.	
Relational operators	replace different comparison operators (>, >=, <, <=, ==, !=) within the program.	
Conditional operators	Replace, insert and remove conditional operators (&&, ||, !). Bitwise operators &, |, and ^ are also used as replacements for these operators as they are very common mistakes.	
Shift operators	Replace bit-wise shifting operators (<<, >>, >>>)	
Bitwise operators	Replace, add and move four operators to perform bitwise functions(&, |, ^, ~).	
Assignment operators	Replace the convenience assignment operators provided by Java with another (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=).	

Table 4.1: Method level mutation operators in μJava. The last column shows a tree diagram where the child nodes are possible mutants generated by this operation applied on the parent.

These method-level mutation operators, presented in Table 4.1 are very generic

and can be applied to other languages with their respective operator sets. μJava also uses class mutation operators, which target feature specific to object oriented languages such as Java. These operators mostly affect how different classes interact with each other, while the method-level mutation operators worked within a single class.

Encapsulation operator set consists of single class *access modifier change*, which means interchanging the four access modifiers `public`, `protected`, *default*, `private` with each other. This is meant to encourage writing test cases that ensure the level of accessibility of methods, classes and class variables is correct. If these test cases are not implemented, this can be a major source of equivalent mutants. Many of these mutations will result in compilation failures and will be detected without executing the test suite.

Inheritance operators cover wide set of mutations relating to the incorrect use of inheritance, e.g. variable shadowing, method overriding, constructors and `super()`, as illustrated in Listing 4.4.

```
public class Parent {
    protected int a;
    public Parent() {
        this.a = 1;
    }
    public int getA() { return a; }
}
public class Child extends Parent {
    int a; // Mutation: Removing this will remove
           // the variable shadowing
    int b;
    public Child(int a, int b) {
        super(); // Mutation: Remove this
        this.a = a;
        this.b = b;
    }
    // Mutation: Remove overriding method
    public int getA() { return a; }
}
```

Listing 4.4: Possible inheritance related mutation operators

Polymorphism is mechanism that allows object references to have different types with different executions. These operators include for example replacing a constructor call with that of a child class (e.g. when `B` extends `A`, `A a = new A(..)` becomes

`A a = new B(..).`) and inserting cast operators (e.g. when `B` extends `A`, `A a = new B(..)` becomes `A a = (A) new B(..).`).

Java-specific features are also covered. These include removing and adding `this` keyword, removing and adding `static` keyword, removing user defined default constructor, member variable initialization deletion, etc.

Bytecode level mutant generation

Since we're focusing our thesis on the Java language, we will examine bytecode level mutant generation in more detail. Java bytecode (or Java virtual machine instructions) is executed on a abstract computing machine called the Java Virtual Machine. (Lindholm & Yellin, 1999) JVM is a virtual machine that handles the details of executing the bytecode on the given platform and the front-end Java compiler doesn't need to take into consideration the target architecture. As an abstract computing machine it provides much of the same functionality as any real computing machine. It has an instruction set which it uses to manipulate its memory. Bytecode is usually produced by the Java compiler, but compilers have been developed for several other languages.

```
 0  iconst_0
 1  istore_1 [i]
 2  invokestatic
    doSomething() : void [17]
 5  iload_1 [i]
 6  iconst_5
 7  if_icmplt 13
10  goto 19
13  iinc 1 1 [i]
16  goto 2
```

Listing 4.5: Original bytecode

```
 0  iconst_0
 1  istore_1 [i]
 2  invokestatic
    doSomething() : void [17]
 5  iload_1 [i]
 6  iconst_5
 7  if_icmpne 13
10  goto 19
13  iinc 1 1 [i]
16  goto 2
```

Listing 4.6: Equivalent mutated bytecode

Listings 4.5 and 4.6 show the relevant parts of the bytecode of the Java program snippets in Listings 4.7 and 4.8 respectively as output by `javap`, Java class file disassembler. Only difference is the opcode in line 7 changing from *"Branch if integer comparison less than"* to *"Branch if integer comparison not equal"*. As the example demonstrates some mutation operations are trivial to implement on the bytecode, and will result in huge computation time savings in program compilation when generating the mutants.

As an example of simple and effective bytecode mutation operators that have

been implemented, we examine Javalanche (Schuler & Zeller, 2009)): It replaces numerical constants ($x \rightarrow x + 1|x - 1|0|1$), negates jump conditions, omits method calls and replaces arithmetic operators. There are no advanced mutation operators related to accessibility, inheritance and polymorphism as μJava has.

Coupling effect

All the mutation operators are very simple and are only applied one at a time. The question remain do they detect complex faults?

> "Test data that distinguishes all programs differing from correct one by only simple errors is so sensitive that it also distinguishes more complex errors." (DeMillo *et al.* , 1978)

Coupling effect has been demonstrated experimentally as well theoretically. (Offutt, 1992) Their results support the idea while testing for simple faults we're also implicitly testing for more complex faults.

4.1.2 Mutation equivalence

One of the problems of mutation analysis is that in practice many mutations produce programs that function identically to the original program. These programs always produce the same output and cannot be killed by adding a new test case. This class of mutants is called *equivalent mutants*, and they are a major reason why mutation analysis has not enjoyed more success in the industry. Listings 4.7 and 4.8 demonstrate the original and an equivalent mutant produced from it.

```
for(int i = 0; ; i++) {
    doSomething();
    if(i >= 5)
        break;
}
```

Listing 4.7: Original

```
for(int i = 0; ; i++) {
    doSomething();
    if(i == 5)
        break;
}
```

Listing 4.8: Equivalent mutant

It is not possible to write a test case to distinguish between the two. It has been shown the recognition of equivalent programs is undecidable. (Budd & Angluin, 1982) This means that in order to completely eliminate equivalent mutants to reach a perfect mutation score, each equivalent mutant must be inspected by a programmer by hand, which can costly and slow.

4.1.3 Mutant elimination

Mutants are killed if they are detected as faulty by the test suite. The test suite only needs to be run until a fault is detected. Coverage data can be gathered on each individual test so that they are only executed if they visit the affected part of the code.

After elimination the dead mutants serve no other purpose than to demonstrate the quality of the test suite. Unlike live mutants they can't be used to generate better tests as they are already detected as faulty.

4.2 Efficient mutation analysis

Even though the idea of mutation analysis was discovered almost 40 years ago, it hasn't been widely adopted. Obvious reason for this is the usage of computation resources. Lack of economic incentive for advanced testing and lack of automatic tools to support mutation analysis have also been speculated as possible reasons. (Wong, 2001) For each mutant a test suite must be executed until a failure is detected or it passes. If the test suite passes, the mutants remains alive and should be examined by the developer, which requires human effort.

How can mutation testing be enhanced in order to make it practical and cost-effective in industry use? Approaches can be divided into roughly three categories, *do fewer*, *do smarter*, and *do faster*. (Wong, 2001) *Do fewer* means generating and analyzing less mutants, while losing as little effectiveness as possible. *Do smarter* approaches concentrate on distributing the workload on several machines. *Do faster* approaches are about generating running and running each mutant as quickly as possible.

4.2.1 Partial Equivalent Mutant Detection

Even though recognizing two programs as equivalent has been shown to be undecidable in general, partial solution can sometimes be found efficiently.

Most obvious class of equivalent mutants is found when mutant has been generated from a compilable language and the resulting binary is identical with the binary from the unmutated source.

4.2.2 Selective mutation

Different mutation operators are not equally good at finding faults per mutation generated. Proper mutation operation selection can reduce the amount of generated

```
final int a = 0;
...
if(a > 0) {
    // If the mutation occurs here it shouldn't
    // affect the resulting bytecode.
}
```

Listing 4.9: Mutation in dead part of the code

mutants from quadratic to linear in the data references, while losing little mutation adequacy. An extensive study was done on Mothra, a mutation test tool for Fortran. Of the original 22 mutation operators they found that a subset of five operators was sufficient for effective mutation testing. (Offutt *et al.* , 1996)

Also different mutation operators will produce varying amounts of mutants. Consider mutation operator that replaces integer constants in the program. On a 32-bit system single all possible values for a single constant would generate 2^{32} mutants, which is obviously unmanageable. Using a better value selection strategy, e.g. such as the one presented in Section 4.1.1 (replacing numerical constants $x \rightarrow x + 1|x - 1|0|1$), will yield fewer mutants as well as fewer equivalent mutants.

4.2.3 Mutant sampling

Instead of examining all the generated mutants, examine only a randomly sampled portion. This isn't very effective as the test set matures, as the number of equivalent mutants remains constant and it becomes harder to find non-equivalent live mutants. This is the simplest form of *do fewer* method, where you indiscriminately and arbitrarily decide not to process certain mutants.

4.2.4 Weak mutation

Weak mutation stops the mutant execution when the mutated portion of the code is executed and examines the interval state of the program with the original. This was implemented in Mothra, where the program state was recorded at the end of every program code block.

4.2.5 Coverage data

Gather coverage data for each test in test suite. When examining the mutant, we only need to run the tests that visit parts of the code affected by this mutation. Because only a single part of the program is modified in a single mutant, the muta-

tion cannot affect the behavior of tests that don't execute it. The coverage data can be gathered at any level of granularity, ranging from classes and packages to single statements in the program. This has yielded huge savings when running mutation analysis on real large-scale projects.

4.2.6 Checking invariant violations

A fundamental problem of mutation analysis is that the most useful (undetected non-equivalent mutants) and the most useless (equivalent mutants) end up in the same category (live mutants). To make things worse, the ratio of equivalent mutants to non-equivalent undetected mutants becomes larger as the test suite gets better and more mature, which makes the interesting mutants even harder to find. Schuler *et al.* , 2009 developed a way to rank this set of mutants according to which are most likely not equivalent mutants. They did it using *dynamic invariants*. Invariants in computer science are almost as old as programming itself. Dynamic invariants are discovered by instrumenting the program and examining its behavior in runtime. (Ernst, 2001) The dynamic invariants are generated for all the methods that are covered by the test suite. When the test set is run on the mutant, we keep track of which of the generated invariants are violated. They use this as approximation of the *impact* the mutation has on the system.

Mutants that violate more invariants are more likely to be non-equivalent than the ones that don't. They also find that a significant portion of the mutants that don't violate any invariant are equivalent mutants. The developer can focus his efforts on the live mutants that violate the most dynamic variants.

Chapter 5

Mutation testing and test suite assessment

Because the focus of this work was the Java programming language, a mutation tool for Java was needed. We go through the how the tool selection process was done, describe the chosen tool, and how it was used to accomplish the goals. We also discuss how the chosen tool could be integrated into CAA workflow.

5.1 Tool selection

Two major freely available mutation analysis tools for Java are μJava and Javalanche. The most significant difference between the two, is that Javalanche generates the mutants from bytecode, whereas μJava uses the Java source code. μJava offers a more versatile mutation operation set, where as Javalanche uses a minimal operation set in order to make it usable in large-scale projects.

No detailed or systematic comparison between the frameworks was done. Javalanche had been reportedly used to successfully run mutation analysis on AspectJ, a very large open source Java project (almost 100 thousand lines of code) in under six hours on a single workstation. (Schuler & Zeller, 2009) No similar large scale use stories were found on μJava. This made us hopeful that Javalanche would probably be able to offer the level of performance required to run mutation analysis in our case, given that the exercises are usually relatively small, but numerous. Given that there were no comparative performance reports on the other tools and the time constraints of this thesis work, Javalanche was chosen, without performing a thorough comparison of the alternatives.

5.2 Javalanche

The coursework we wanted to analyze was written in Java, so a mutation analysis tool for Java was needed. Javalanche was chosen as the mutation analysis implementation, because of its efficiency and proven track record on analyzing large open source software components. To explore the usefulness of mutation analysis in course work grading, we used Javalanche to calculate mutation scores on actual coursework.

In order to perform code coverage on a test set, we need the program and the test set. To perform mutation analysis we have stricter requirements:

1. The program and the test set must compile successfully.
2. The test suite must pass on the unmutated program.
3. The individual tests must be repeatable and independent of the execution order.

Javalanche goes through several phases during the analysis to check if these requirements are met. First the program and the test suite are compiled and the test is run. If any of the tests fail, it will not proceed. Next the javalanche executes the tests multiple times in different order and examines the results. This is done to test if test suite is implemented in a way that the tests are independent of each other and the order of execution. It can uncover erratic tests (tests that may sometimes fail and pass). Failure here will also abort the task. It's common for unit tests to be executed only once during the lifetime of the virtual machine. Javalanche executes the same test suite multiple times, which means that the unit tests must have proper **setUp()** and **teardown()** methods, instead of relying on the class loader, as exemplified in listings 5.1 and 5.2. In Listing 5.1 Fixture **f** is initialized once when the class is loaded by class loader, and if **testF()** is run multiple times, the same instance of **f** is used. In Listing 5.2 fixture **f** is reinitialized in the **@before** hook, before running the test, which means the test can be run multiple time, and it will always use a fresh instance of the fixture.

After the test suite has passed these initial steps, the actual mutation analysis can begin. First the program is scanned for the classes to be mutated. This usually will require developer help, since we usually want to restrict the generation of mutants to the portion of the code tested by the student, and other auxiliary classes should be excluded. Javalanche requires that all the classes are in a explicit package, instead of the default package. Then the mutants themselves are generated, executed and

```
private Fixture f =
    new Fixture();

@Test
public void testF() {
    ...
}
```

Listing 5.1: Non-repeatable test class

```
@Before public void setUp() {
    f = new Fixture();
}
@Test
public void testF() {
    ...
}
```

Listing 5.2: Test class with proper setup.

finally the results are summarized. We're mostly interested in the mutation and using it for grading purposes, but the individual live mutants may be examined at this point also.

The whole process is done in a single Ant script, with different phases of the process as different targets. A relational database system (by default HSQLDB[1]) is used to store the intermediary and end results, such as individual mutants, and the results of its test set run. There is support for distributing the work across multiple cores/processors.

Javalanche is by no means a stand-alone project. The significant external components of Javalanche include:

- **asm** is a general purpose bytecode manipulation and analysis framework. It is essential for mutant generation. (`http://asm.ow2.org/`)
- **daikon** is a dynamic invariant detector. Used to assess the impact of individual live mutants. (`http://groups.csail.mit.edu/pag/daikon/`)

5.3 How Javalanche was used

Running Javalanche on the actual course work is relatively straightforward. Javalanche requires that the mutated classes have a package, instead of the default package, which means that the **package** declaration must be augmented into the source files if it is missing. Exercise specific exclusion list is also needed, in order to restrict the mutations to the part of the code that the student is expected to test. After the exclusion list is defined and the program compiled running Javalanche is straightforward. Standard output (**stdout**) is redirected to a log file, from which the mutation analysis results are extracted afterwards.

[1] `http://hsqldb.org/`

Sometimes the mutation analysis cannot be successfully completed. This is most likely caused by the software failing to comply with requirements listed in Section 5.2. Javalanche is a relatively new project with complex functionality, and it seems to fail inexplicably with non-informative error messages, but this should remedy itself as the project matures.

In order to use Javalanche with a CAA system the following things need to be done:

1. Program **P** and test suite **T** must input to Javalanche.
2. Parts used in the mutation generation in **P** must be defined. The portion of **T** used in testing must be defined. These are both done using respective exclusion lists.
3. Running Javalanche.
4. Extracting the interesting information from the Javalanche's output.

The first phase is simple. Compiling the program and adding its class path into Javalanche's configuration is enough.

The exclusion lists are found in `mutation-files/`, where there are files `exclude.txt` (class exclusion) and `test-exclude.txt` (test exclusion). The actual content of these lists should depend on the specific exercise, and can be pregenerated if the students are expected to only work on predefined set of class definitions.

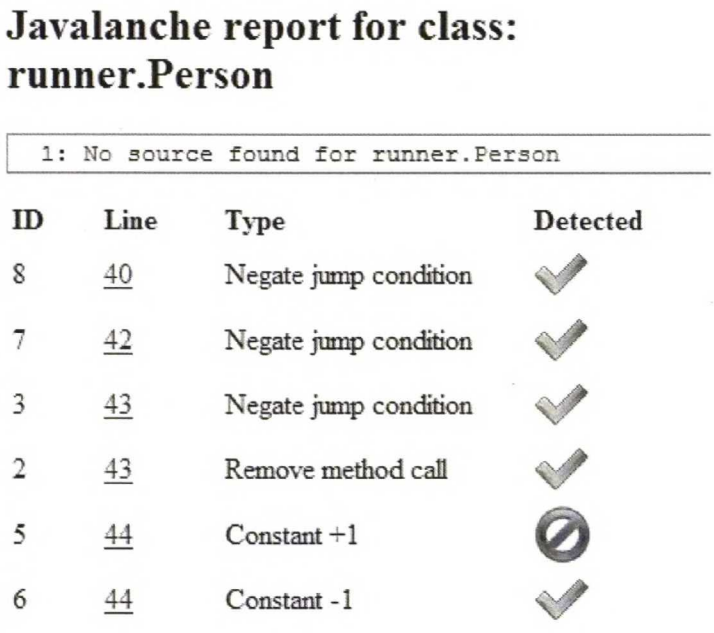
Javalanche is then run as an ant script. It requires that Javalanche is installed, and that there's a JRE (Java Runtime Environment) available which the user can run.

```
Total mutations:                                    190
Touched mutations:                                  185 (97,37%)
Not touched mutations:                                5 ( 2,63%)
Killed mutations:                                   126 (66,32%)
Survived mutations:                                  64 (33,68%)
Mutation score:                                  66,32%
Mutation score for mutations that were covered:  68,11%
```

Listing 5.3: Example Javalanche mutation analysis summary report

Typical Javalanche mutation analysis report is seen in Listing 5.3, it can be found in `mutation-files/report`. Here you can find all the quantitative information that can be used in the assessment. The most interesting of these is the total number of mutants generated, examined, and killed, as well as mutation score itself.

Javalanche also generates a list of the individual mutants generated per class. Example of such report is seen in Figure 5.1. Possible uses of this information are discussed in Chapter 7.



Javalanche report for class: runner.Person

```
1: No source found for runner.Person
```

ID	Line	Type	Detected
8	40	Negate jump condition	✓
7	42	Negate jump condition	✓
3	43	Negate jump condition	✓
2	43	Remove method call	✓
5	44	Constant +1	⊘
6	44	Constant -1	✓

Figure 5.1: Javalanche class mutation report.

5.4 Observations on Javalanche

Javalanche presents several implementation specific issues. Since mutant exclusion is defined on the class level, the student generated code should be confined to their own classes in order to minimize mutants generated for the part of program that is untested by the student.

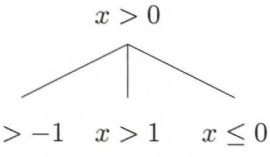
5.4.1 Methods with boolean return values

Javalanche mutation set generates unnecessary mutants for boolean constant statements/methods, i.e. `return true` generates 3 mutants. All the equivalent mutants found by hand were caused by this. Consider the following Java method in Listing 5.4.

```
public static boolean greaterThanZero(int n) {
    return n > 0;
}
```

Listing 5.4: Boolean method example.

One would expect Javalanche to generate the following three mutants:



Instead surprisingly five mutants are generated. This is caused by how Java boolean methods are compiled into bytecode, as illustrated in listing 5.5.

```
public static boolean greaterThanZero(int);
   0:   iload_0      // push the method argument into the stack
   1:   ifle     6   // pop the stack and jump to 6 if it's
                     // less than or equal to 0
                     // mutant => ifgt  6
   4:   iconst_1     // push constant 1 into the stack
                     // mutants => iconst_0 and iconst_2
   5:   ireturn      // return 1
   6:   iconst_0     // push constant 0 into the stack
                     // mutants => iconst_1 and iconst_m1 (-1)
   7:   ireturn      // return 0
```

Listing 5.5: One possible disassembled compilation of **greaterThanZero(int n)**. Javalanche would generate the mutants from lines 1, 4 and 6. The problematic mutant is generated in the line 4 (**i_const_2**). It causes the method to return 2, and JVM handles boolean values as integers where zero is **false** and non-zero value is **true**.

It should be possible to alter Javalanche to take into account these situations. The mutation operator could work in differently if the following conditions are met:

1. If the instruction is **iconst_0** or **iconst_1**.
2. The current method has boolean return value.
3. The following instruction is integer return statement **ireturn**.

If these conditions are met, only one mutant should be generated (**iconst_0** for **iconst_1** and vice versa).

Chapter 6

Results and analysis

We use actual student generated coursework to assess the potential of using mutation analysis in automatic assessment. The test data is from Helsinki University of Technology's course called *Intermediate Course in Programming T1*[1] held in spring 2009. The course teaches the basics of object oriented programming and the Java programming language. The course is worth 6 ECTS-credits. It is directed at computer science majors, and before taking this course, the students are expected to complete a basic programming course worth 5 ECTS-credits, which also teaches Java.

The course uses Web-CAT as a CAA system. The students were also instructed to provide a test suite for their implementation, and they were rewarded for them based on the statement coverage.

Section 6.1 presents the test sets we will use, and presents the results of quantitative analysis done on it. Section 6.2 examines individual samples and discusses what kind of assessment and feedback can be derived from the results.

6.1 Test sets and quantitative analysis

Three different programming exercises were used as the test sets, the first dealing with binary search trees (Subsection 6.1.1), the second with hash tables, and different hashing schemes (Subsection 6.1.2), and the third with graph algorithms (Subsection 6.1.3).

Mutation analysis was performed on all the individual submissions in the test sets. These results were compared with the assessment done by the current CAA system. We are mostly interested in finding test suites that are weak according to

[1] `https://noppa.tkk.fi/noppa/kurssi/t-106.1240/etusivu`

mutation analysis, but have been given a good score by the previous system.

For the Test set A, we will perform a cross comparison trying to determine how much the differences between the implementations affects the mutation score.

6.1.1 Test set A - Binary search trees

In the first exercise, the students were tasked with implementing three basic operations for the binary tree data structure[2]:

1. Adding a node
2. Searching the tree for a given key and returning the node
3. Printing the result of the inorder tree traversal

In addition to implementing these operations, the students were also expected to provide unit tests for them. The grading of the test suites was based on the code coverage.

We ran mutation analysis using Javalanche on all the students' final submission ($N = 161$). Mutation analysis was successful on 131 items (81.3%). Chapter 5 discusses the various reasons why mutation analysis can be unsuccessful even if the test suite can be executed successfully.

Of the 131 analyzed submissions 125 achieved perfect code coverage. The spread of the code coverages is illustrated in Figure 6.1. If we used code coverage as a metric for the test quality, the result would be excellent. This is an expected result, as the students were awarded for reaching perfect coverage, and in the case of this assignment it was relatively easy to achieve; a single non-degenerate[3] tree should suffice as the test data.

The mutation scores are spread over a wider range, as illustrated in Figure 6.2. The mutation analysis yielded an average of 44 mutations per sample, and it took on average about 12 seconds to run per sample. The best work managed to kill 48 of its 49 mutants, resulting in 97.96% mutation score. The one remaining live mutant was identical on the java source code level and thus unkillable, so this can be considered a perfect score. On average the mutation score was 80.48%, and the worst was 40%. The worst mutation score that had reached perfect code coverage was 54.76%. There were several samples, where the tested method could be

[2]The exercise instructions can be found at `http://www.cs.hut.fi/Opinnot/T-106.1240/2008_external/harjoitukset/kierros_1/harj_4/index.html` (in finnish).

[3]In a degenerate binary tree, each node has at most a one child, and thus behaves as a linked list.

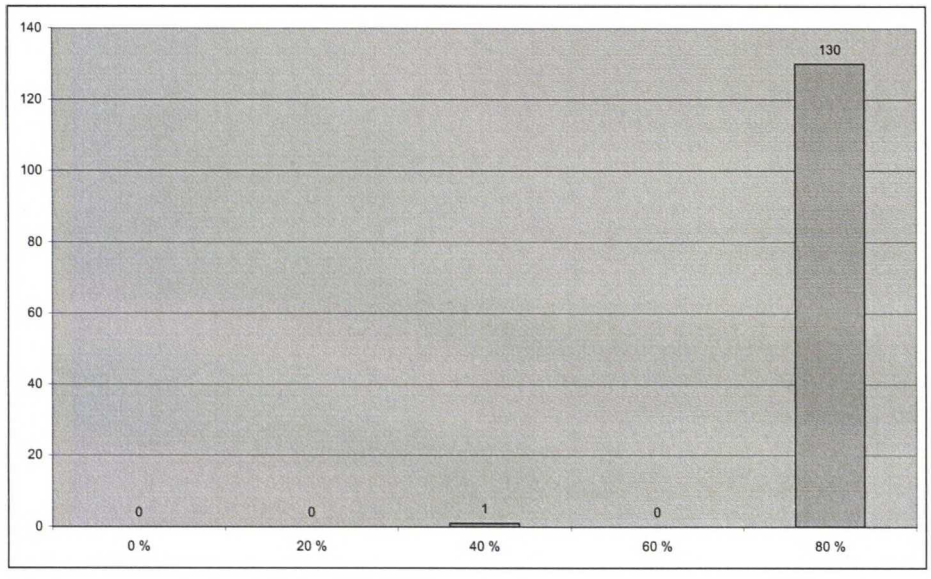


Figure 6.1: Code coverage spread of the test set A

completely commented out, and the test suite would fail to detect this. This would seem to support our hypothesis that mutation analysis offers superior capabilities in identifying weak test sets.

Figure 6.3 illustrates relationship between mutation score and test coverage in the set as a scatter plot. Histograms on each axis show the distribution of the respective variables. Code coverage being on the X-axis causes the data points to be clustered on right edge of the graph as the code coverage was 100% for most the submissions. Casual observation of the graph would seem to indicate that there does not seem to be a correlation between the variables. Pearson product-moment correlation[4] coefficient for the dataset is $\rho \approx 0.1628$ indicating a very small or nonexistent positive correlation.

[4]Pearson product-moment correlation is defined as:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\mu_X \mu_Y}$$

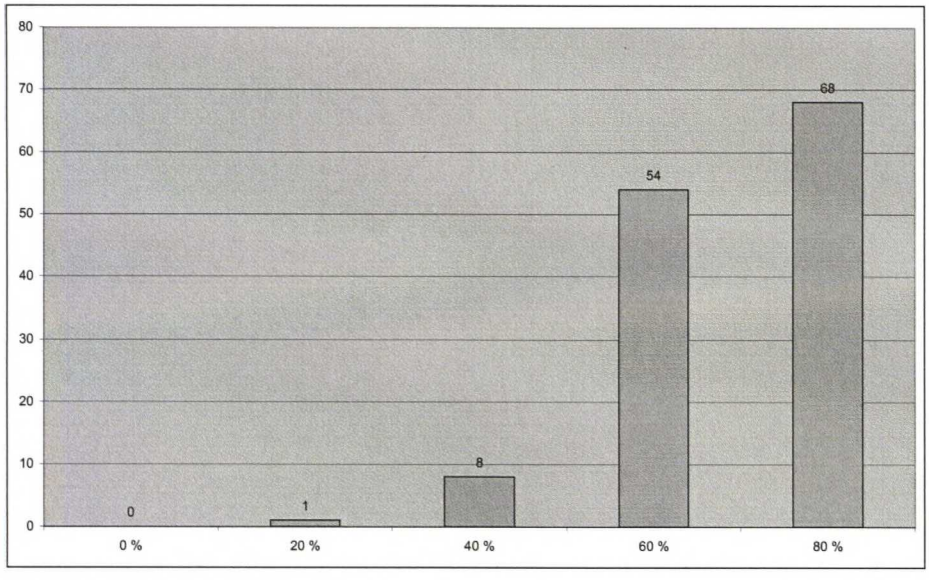


Figure 6.2: Mutation score spread of the test set A

In order to show how much variation does the implementation itself (excluding the tests) causes to the mutation score, we selected 4 example test suites from the ones that had achieved 100% test coverage; the worst, the best, and two others. We ran mutation analysis on all the previously analyzed submissions with each of these test sets, and results are shown in Figure 6.4.

If we are to use the mutation score as an indication to the adequateness of the test set, this score should not be affected by the implementation, but the implementation does affect the number of equivalent mutants created, which makes the mutation scores of two test suites on two different implementations incomparable. It should be noted that in Figure 6.4 the distribution of the first two test sets seems to be very similar even though the original score is very different. This is something that needs to taken into account if the students are ever rewarded on basis of its mutation score.

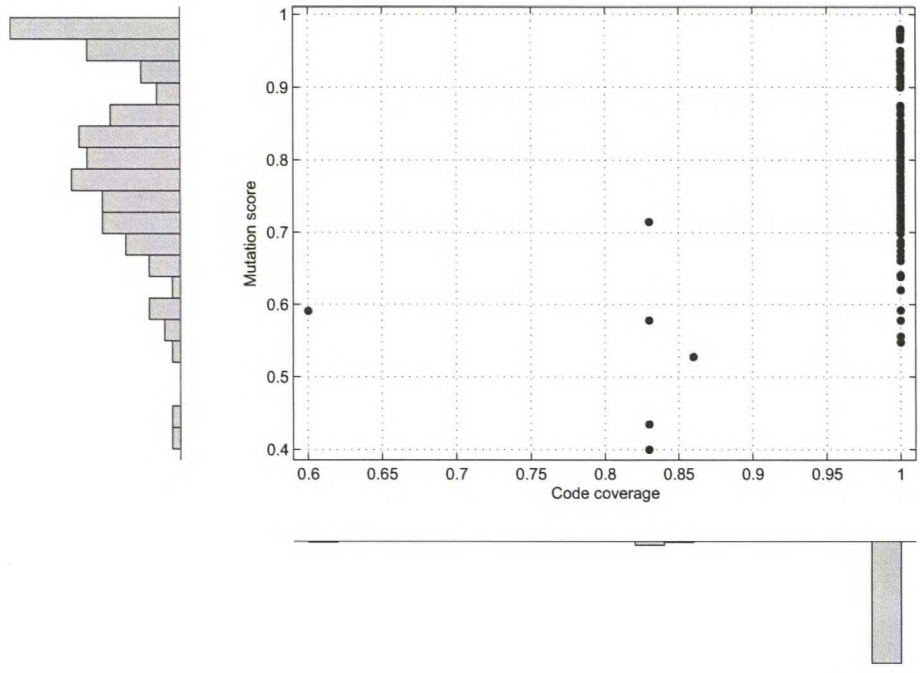


Figure 6.3: Scatter plot of code coverage and mutation score of the test set A

6.1.2 Test set B - Hashing

In the second assignment, the students two tasks related to a hash table implementation:

1. Implement **nextPrime(int n)**, that returns the smallest prime number that larger than **2*n**. It is used to calculate the new size of the underlying array, when it is getting full.

2. Thoroughly test the rest of the hash table implementation, provided with the assignment.

The hash table grows dynamically and uses a two phase hashing to calculate its position in the array. We ran mutation analysis using Javalanche on all the students' final submission ($N = 181$). Mutation analysis was successful on 174 items (96.1%).

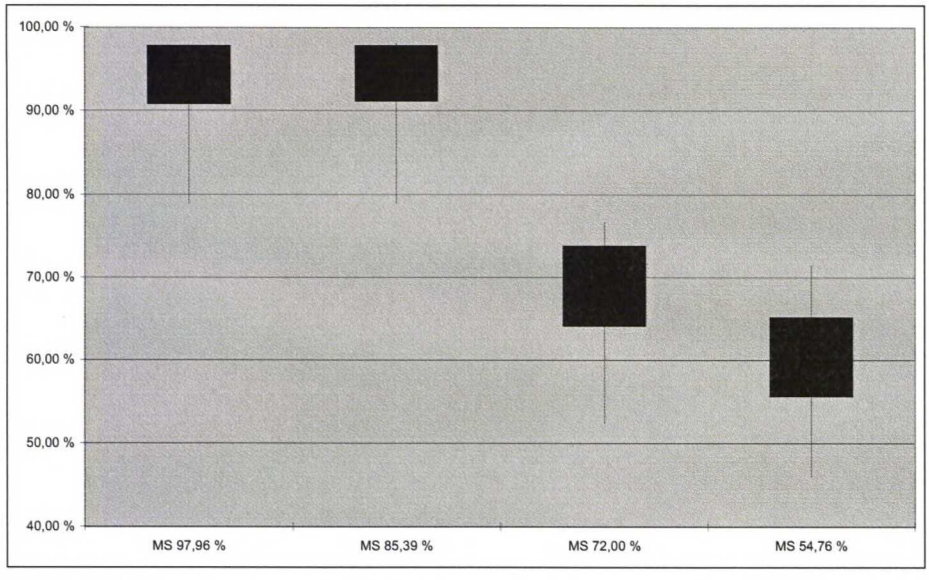


Figure 6.4: Distribution of mutation scores for the selected test suites as box plot. Displayed are the minimum, maximum 90th and 10th percentiles. X-axis labels are the mutation scores that were achieved when running the mutation analysis on its respective implementation.

Of the 174 analyzed submissions 135 achieved perfect code coverage. This exercise was more complex than test set A, as it yielded on average 106 mutations per sample. Mutation scores of the submissions that achieved perfect coverage score ranged from 42.7% to 89.39% with 73.73% being the average. Mutation scores of the submissions that didn't reach perfect coverage ranged from 24.44% to 85.96% with 63.52% being the average.

The distribution and the relationship between code coverages and mutation scores can be seen in Figure 6.5.

The best sample reached 100% code coverage and managed to kill 100 out of 112 mutants. We were able to augment this test suite by adding new tests in a way that 111 of the 112 mutants were killed, with the one remaining mutant being an equivalent mutant, thus reaching a mutation score of 99.11%, which is the perfect mutation score for this implementation.

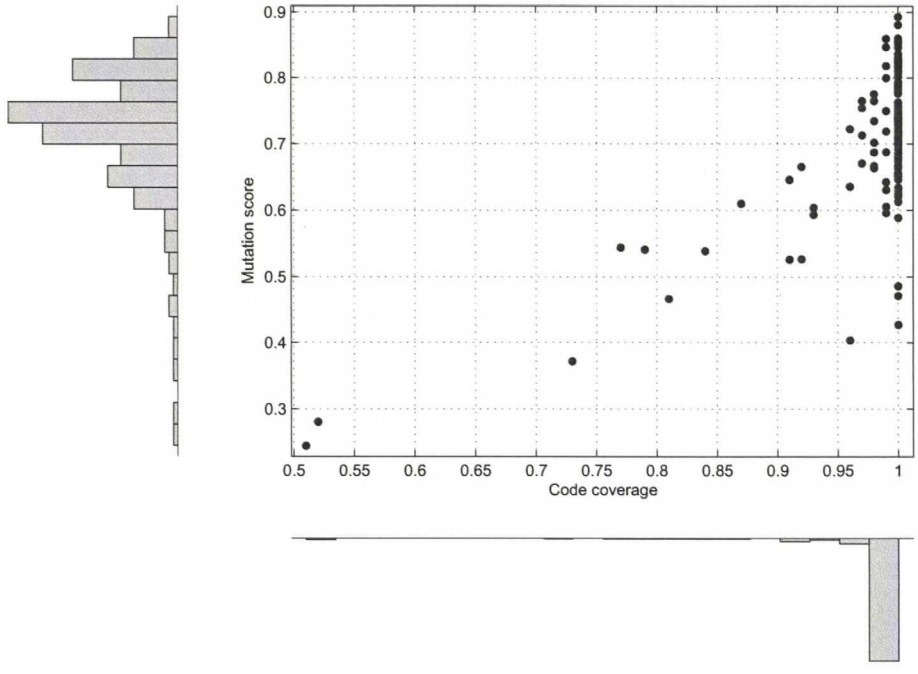


Figure 6.5: Scatter plot of code coverage and mutation score of the test set B

We also examined the sample with the worst mutation score that had reached perfect code coverage: The sample managed to kill 38 of the 89 total mutations, reaching a mutation score of 42.70%. The student generated portion of the test suite had only a single assertion, and half of the unit tests didn't contain any assertions. The test suite is clearly inadequate despite it having perfect branch coverage.

The distribution seems to be very similar to the distribution seen in test set A (Figure 6.3), except that the number of samples is higher and there is more variance in the code coverages, $\sigma \approx 0,064$. Pearson product-moment correlation coefficient for the dataset is $\rho \approx 0,669$ indicating a clear positive correlation, unlike in test set A. Major difference with test set A is the maximum mutation score, which was under 90% compared to the practically perfect mutation score achieved in A. Not even the best student generated test suite managed to kill all the non-equivalent mutants.

6.1.3 Test set C - Disjoint sets

The students were tasked with implementing a data structure for disjoint sets, under the guise of social networks:

1. Implement method **meet(Person)**, which joins two sets if they are disjoint
2. Implement method **knows(Person)**, which queries whether or not the two nodes belong to the same set.
3. Generate a test suite for this class.

Of the 193 analyzed submissions mutation analysis was successfully performed on 169 submissions, of which 144 achieved perfect code coverage. Each submission yielded on average 26 mutations. The amount of generated mutants ranged from 12 to 93. Mutation scores of the submissions that achieved perfect coverage score ranged from 38.46% to 95.00% with 84.88% being the average, which is the highest in all the data sets. Mutation scores of the submissions that didn't reach perfect coverage ranged from 21.43% to 90.48% with 67,84% being the average.

The distribution and the relationship between code coverages and mutation scores can be seen in Figure 6.6.

The best sample had 100% code coverage and killed 19 of its 20 mutations reaching a mutation score of 95.00%, and the remaining mutant was equivalent so this sample should considered mutation adequate.

The submission with perfect code coverage and worst mutation score managed to kill 10 of its 26 total mutations reaching a mutation score of 38.46%. We were able to augment this test suite to kill 22 of its 26 total mutation reaching mutation score of 84.62%, with the 4 remaining mutants being equivalent.

The distribution seems to be very similar to the distribution seen in the previous test sets (Figure 6.3 and 6.5). Pearson product-moment correlation coefficient for the dataset is $\rho \approx 0.6034$ indicating a clear positive correlation.

6.2 Qualitative analysis

In this section we will do qualitative analysis on the individual samples in the test sets. This examination is focused in the samples that have reached perfect code coverage, but have received the poorest mutation score. If mutation analysis offers a superior test adequacy criterion, these test suites should be of poor quality by manual inspection.

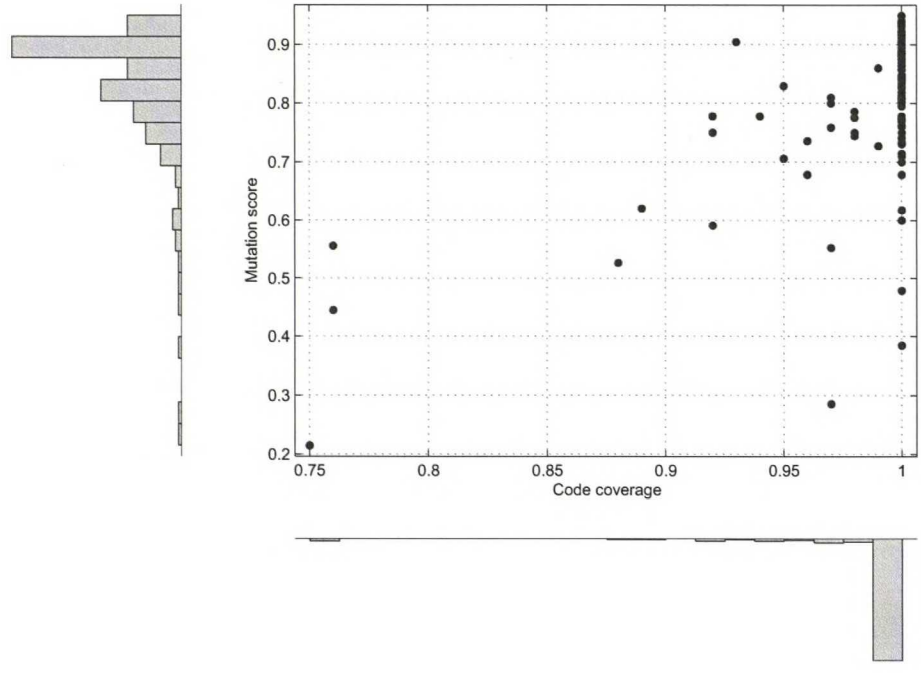


Figure 6.6: Scatter plot of code coverage and mutation score of the test set C

We examined bottom four samples by mutation of score of the samples that had reached perfect coverage. We try to identify poorly tested and untested functionality. Other aspects of test quality, such as style and structural consideration are outside the scope of this examination.

6.2.1 Test set A

First examined sample detected 23 of 42 generated mutants resulting in mutation score of 54.76%. Web-CAT gave it 57% problem coverage. Following observations were made:

1. The method **printInorder(PrintWriter pw)** is completely untested. In fact you can comment its implementation out and it still won't be detected by the test suite.

2. Methods **contains(ItemType data)** and **insert(ItemType data)** both

contain errors.

The second sample detected 30 of 54 generated mutants resulting in mutation score of 55,56%. Web-CAT gave it 86% problem coverage. Following observations were made:

1. The method **printInorder(PrintWriter pw)** is completely untested, as in the first sample.
2. Inserting a node to the left-hand side of a node is untested also.
3. Method **contains(ItemType data)** if the searched data is in the root node. The method also has lots of redundant code, which bloats the number of equivalent mutants.

The third sample detected 26 of 45 generated mutants resulting in mutation score of 57.78%. Web-CAT gave it 71% problem coverage. Following observations were made:

1. Once more the method **printInorder(PrintWriter pw)** is completely untested.
2. Method **insert(ItemType data)** contains an error which sets the root of the tree to point to the last node inserted.

The last sample detected 29 of 49 generated mutants resulting in mutation score of 59.18%. Web-CAT gave it 100% problem coverage. This makes it the most interesting sample to be examined in this test set, because the current system gave it a perfect score on all categories, but the low mutation score indicates poor test quality. Following observations were made:

1. As in all the other samples, the method **printInorder(PrintWriter pw)** is completely untested.
2. Inserting a node to the right hand side of another node is also untested.

Common feature for all four samples is that the method **printInorder(PrintWriter pw)** was completely untested, and the test quality of all these samples was poor. Listing 6.1 illustrates an example test case for the operation.

```
@Test public void testPrintInorder() {
    BinarySearchTree<Integer> testTree =
        new BinarySearchTree<Integer>();
    StringWriter result = new StringWriter();
    testTree.insertItem(2);
    testTree.insertItem(1);
    testTree.insertItem(3);
    testTree.printInorder(new PrintWriter(result));
    /* Code below is optional for perfect coverage. */
    StringWriter expected = new StringWriter();
    PrintWriter p = new PrintWriter(expected);
    p.println(1);
    p.println(2);
    p.println(3);

    assertEquals(expected.toString(), result.toString());
}
```

Listing 6.1: Test case for the print inorder operation. The code below the comment can be omitted and still perfect coverage is reached with most implementations, but nothing is tested about the method's result.

6.2.2 Test set B

The first examined sample detected 38 of 89 generated mutants resulting in mutation score of 42.70%. Web-CAT gave it 100% problem coverage. The test suite contains only one student generated assertion (another three were part of the template the student was tasked with completing). This indicates that extremely minimal testing was done, but a perfect score was achieved. Only the negative case of **contains(Hashable item)** was tested.

The second sample detected 49 of 104 generated mutants resulting in mutation score of 47.12%. Web-CAT gave it 100% problem coverage. It is very similar to the first sample, as it only contains one student generated assertion (whether or not the first insertion with a given hash code is placed in the correct position in the underlying array).

The third sample detected 52 of 107 generated mutants resulting in mutation score of 48.60%. Web-CAT gave it 100% problem coverage. Student generated portion of the test suite, consisted of one gigantic test, with a single assertion.

The last sample detected 53 of 90 generated mutants resulting in mutation score of 58.89%. Web-CAT gave it 100% problem coverage. Only inserting a single key and delete operation were tested.

All the samples contained plenty of untested functionality, and are considered poor by manual inspection.

6.2.3 Test set C

The first examined sample detected 10 of 26 generated mutants resulting in mutation score of 38.46%. Web-CAT gave it 100% problem coverage. Following observations were made:

1. Return values of **meet(Person)** were untested.
2. Behavior of **meet(Person)** when the two persons already know each other was untested.
3. Redundant code in **meet(Person)**, which caused several equivalent mutants.

The second sample detected 11 of 23 generated mutants resulting in mutation score of 47.83%. Web-CAT gave it 100% problem coverage. Only **toString()** method is tested. Interestingly this sample scored higher than the first sample, even though the test suite was inferior.

The third sample detected 15 of 25 generated mutants resulting in mutation score of 60.00%. Web-CAT gave it 86% problem coverage. Following observations were made:

1. Method **know(Person)** is overly complex, and contains redundant code, which inflates the amount of equivalent mutants.
2. Method **know(Person)** is not tested to return **true** in the event that a link is created between two disjoint sets.

This is the first sample analyzed with poor mutation score that has a very decent test quality on manual inspection. The amount of generated equivalent mutants is high because of implementation specific issues.

The last sample detected 13 of 21 generated mutants resulting in mutation score 61.76% Web-CAT gave it 100% problem coverage. Only problem found in the test suite was that method **know(Person)** was untested when the target node was not the root node. Overall the test quality was very good. Redundant code in the implementation caused a high number of equivalent mutants again.

Unlike in the other test sets, the samples in the test set C were not all of bad quality.

6.3 Summary

Mutation analysis was performed on three different test sets, as summarized in Table 6.1. Columns from left right are name of the test set, amount of samples on which Javalanche was successfully run on, the amount of mutants Javalanche generated, and amount of samples examined with either good or bad test quality.

	Mutation analysis			Generated Mutants			Test quality count	
Name	succ	fail	%	min	max	avg	bad	good
Set A	131	27	83.0%	22	90	46.4	4	0
Set B	174	13	90.0%	79	439	106.9	4	0
Set C	169	24	87.6%	12	93	26.4	2	2

Table 6.1: Summary of the test sets used.

Qualitative analysis performed in Section 6.2 clearly showed that the test suites with poor mutation score contained serious shortcoming, except in the case of Set C. Because the assignment was so simple (very few mutants generated), and easy to test, the equivalent mutants skewed the results in the two cases.

The equivalent mutants examined can be split into two categories. First category isthe equivalent mutants created by Javalanche, mostly from methods returning boolean values as described in Subsection 5.4.1. Second category are caused by the implementation itself (redundant or dead code).

Chapter 7

Conclusions

In this chapter we will discuss the findings of this work. Section 7.1 answers the research questions posed in the Introduction. Section 7.2 discusses possible subjects for further research.

7.1 Answers to the research questions

We recall the research questions we set out to answer:

Q1: Can mutation analysis be used to give meaningful feedback and grading on test suites included in programming assignment submissions?

The qualitative analysis done supports the notion that test suites with perfect code coverages and poor mutation scores are often of poor quality. One of the test sets indicated that if the software is very simple and few mutants are generated, significant portion of the live mutants can be equivalent, which makes mutation analysis less effective in identifying weak test suites in small assignments. In addition, the amount of equivalent mutants generated is implementation dependent, which make it difficult to use the mutation score directly in grading.

If the CAA system were to use mutation analysis to grade the test quality, the student might not realize why he is being penalized. Code coverage as a metric is simpler to understand, where as the mutation analysis process is much more complex. To alleviate this problem other kind of feedback should be derived from the mutation analysis besides the mutation score and the number of live mutants. The specific methods where the live mutants occur could be used as tips to the students on how to improve their test sets. The methods that have no live mutants left are mutation adequate, and the student can be instructed to focus his efforts

elsewhere.

If the student realizes how his test suite is being graded, he may try to fool the mutation analysis system by seeding irrelevant code into his submission, which is easily testable but yields a huge number of mutants, as illustrated in Listing 7.1. This will distort the mutation score. The large number of analyzed mutants can also result in the grading system performing poorly, which should be taken into account. The number of mutants generated per submission should also be monitored, as it can indicate this kind of cheating, or malicious intent in trying to cause the system to perform poorly.

```
public static int dummy(int x) {
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;x+=3;
    return x;
}
```

Listing 7.1: Sample of a easily testable dummy method, that will yield over 150 mutants with Javalanche, and can be simply tested with `assertEquals(180,dummy(0));`. The method itself is simply performs simply the function $f(x) = x + 180$, but the way it is written and how Javalanche works, each constant and convenience assignment operator generates several mutants.

Mutation analysis cannot always be run successfully. Unlike code coverage calculation, mutation analysis needs a test suite with all assertions passing. In about 12% of the examined samples mutation analysis failed. In some situation Javalanche failed to perform mutation analysis, with no apparent cause. Javalanche is a relatively new project, and these problems should remedy themselves as the project matures.

Coverage analysis should continue to be in conjunction with mutation analysis. While coverage analysis says nothing about how well something has been tested, it can be used to identify completely untested portions of software with relatively little computational cost.

In summary, mutation analysis has potential in CAA, provided that the exercises are not too small or large. If the exercise is very small, it will create few mutants, and equivalent mutants can skew the results heavily, as was seen in Test set C. On the other hand, if the exercises are very large, the assessment can take too much time

because of the large number of mutants, and the computational cost can become prohibitive.

Q2: Can the system presented in **Q1** be scaled to a handle large number of submissions in a course attended by hundreds of students?

In the course of doing the quantitative analysis described in Section 6.1, we ran mutation analysis on hundreds of samples in matter of hours, using a single workstation. Since it's trivial to distribute this workload to several machines if necessary, we don't expect the performance to be a problem with our course and exercise size. This heavily depends on the size of the program and the number of mutants generated, and needs to taken into account when designing the assignments, if mutation analysis is to be used.

Q3: Are there currently tools available that can be used to implement the system presented in **Q1**?

Javalanche was the only framework evaluated, and for our purposes it performed excellently. The amount of generated mutants was lower than anticipated. Only significant problem found were the equivalent mutants generated from boolean returning methods, described in Subsection 5.4.1. Because the mutants are generated in the bytecode level, the sample needs only be compiled once, which reduces the overhead and thus increases performance.

7.2 Future work

7.2.1 Mutation analysis tool for educational use

Even if the mutation scores provided by the analysis aren't used directly to grade the submitted test sets, there may other ways that mutation analysis may be useful in computer science education. One idea is to develop a plug-in for the IDE (Integrated Development Environment). The workflow could be as follows:

1. Select the implementation classes for mutation analysis, and use this information to generate the exclusion lists.
2. Perform mutation analysis based on these lists.
3. Display the live mutants with highest impact, as calculated by Javalanche, and annotate these lines and relevant mutations in the code editor.

The student can then use this tool to improve his test suite. Because mutation analysis is a relatively advanced software testing technique, it may be wise not use such a tool in the introductory programming courses.

Given that the Javalanche is used to perform mutation analysis in Java, Eclipse[1] would seem to be the ideal platform for the tool.

There already exists MuClipse[2] mutation analysis plug-in for Eclipse, which is based on μJava. Before the invention of using invariant violations to rank the live mutants, such a system would have been limited in its usefulness, because of effort required to go through number of equivalent mutants.

It seems that there already is an Eclipse plug-in for Javalanche, but it has not been made available to the public. A screenshot of it can be seen in Schuler & Zeller, 2009. We were unable to obtain it for evaluation when doing this work.

[1] http://www.eclipse.org/
[2] http://muclipse.sourceforge.net/

REFERENCES

Ala-Mutka Kirsti. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, **15**(2), 83–102.

Ala-Mutka Kirsti, & Järvinen H.-M. 2004. Assessment process for programming assignments. *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*, 181–185.

Ala-Mutka Kirsti, Uimonen Toni, & Järvinen Hannu-Matti. 2004. Supporting Students in C++ Programming Courses with Automatic Program Style Assessment. *Journal of Information Technology Education, volume 3*, 245–262.

Antoniol G., Casazza G., Penta M. Di, & Fiutem R. 2001. Object-oriented design patterns recovery. *J. Syst. Softw.*, **59**(2), 181–196.

Budd T. A., & Angluin D. 1982. Two notions of correctness and their relation to testing. *Acta Informatica*, 31–45.

Carter Janet, Ala-Mutka Kirsti, Fuller Ursula, Dick Martin, English John, Fone William, & Sheard Judy. 2003. How shall we assess this? *Pages 107–123 of: ITiCSE-WGR '03: Working group reports from ITiCSE on Innovation and technology in computer science education.* New York, NY, USA: ACM.

DeMillo R. A., Lipton R. J., & Sayward F. G. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, **11**, 34–41.

Dijkstra Edsger W. 1972. Chapter I: Notes on structured programming. 1–82.

Edwards Stephen H. 2003. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, **3**(3), 1.

Edwards Stephen H. 2004. Using software testing to move students from trial-and-error to reflection-in-action. *Pages 26–30 of: SIGCSE '04: Proceedings of the*

35th SIGCSE technical symposium on Computer science education. New York, NY, USA: ACM.

Ernst M.D.; Cockrell J.; Griswold W.G.; Notkin D. 2001. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, **27**, 99–123.

Geist R., Offutt A.J, & Jr. F.C. Harris. 1992. Estimation and enhancement of real-time software reliability through mutation analysis. *Computers, IEEE Transactions on*, **41**, 550–558.

Goodenough John B., & Gerhart Susan L. 1975. Toward a theory of test data selection. *Pages 493–510 of: Proceedings of the international conference on Reliable software.* New York, NY, USA: ACM.

Gosling James, Joy Bill, Steele Guy, & Bracha Gilad. 2005. *The Java Language Specification.* `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`.

Hevery Misko. 2008. Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process. *Pages 747–748 of: OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications.* New York, NY, USA: ACM.

Higgins Colin, Hegazy Tarek, Symeonidis Pavlos, & Tsintsifas Athanasios. 2003. The CourseMarker CBA System: Improvements over Ceilidh. *Education and Information Technologies*, **8**(3), 287–304.

Howles Trudy. 2003. Fostering the growth of a software quality culture. *SIGCSE Bull.*, **35**(2), 45–47.

IEEE 610.12. 1990. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1.

Jackson David, & Usher Michelle. 1997. Grading student programs using ASSYST. *SIGCSE Bull.*, **29**(1), 335–339.

Lindholm Tim, & Yellin Frank. 1999. *The Java™ Virtual Machine Specification.* Second edn. Prentice Hall PTR. `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`, checked October 30th, 2009.

Offutt A. Jefferson. 1992. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, **1**(1), 5–20.

Offutt A. Jefferson, Lee Ammei, Rothermel Gregg, Untch Roland H., & Zapf Christian. 1996. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, **5**(2), 99–118.

Offutt Jeff. 2008 (November). *μJava Home Page.* `http://cs.gmu.edu/~offutt/mujava/`.

Robins Anthony, Rountree Janet, & Rountree Nathan. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*, **13**, 137–172.

Schuler David, & Zeller Andreas. 2009. Javalanche: efficient mutation testing for Java. *Pages 297–298 of: ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium.* New York, NY, USA: ACM.

Schuler David, Dallmeier Valentin, & Zeller Andreas. 2009. Efficient mutation testing by checking invariant violations. *Pages 69–80 of: ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis.* New York, NY, USA: ACM.

van Deursen Arie, Moonen Leon, van den Bergh Alex, & Kok Gerard. 2001. Refactoring Test Code. *In: Proc. Second International Conf. Extreme Programming and Flexible Processes in Software Eng.*

van Rompaey Bart, Du Bois Bart, Demeyer Serge, & Rieger Matthias. 2007. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Trans. Softw. Eng.*, **33**(12), 800–817.

Wong W. Eric. 2001. *Mutation testing for the new century.* Springer.

Zhu Hong, Hall Patrick A. V., & May John H. R. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.*, **29**(4), 366–427.